

MEtaGile: A Pragmatic Domain-Specific Modeling Environment

Olivier Buchwalder, Claude Petitpierre

Networking Laboratory

Swiss Federal Institute of Technology in Lausanne

CH-1015 Lausanne EPFL, Switzerland

E-mail: {olivier.buchwalder, claudette.petitpierre}@epfl.ch

Abstract

Domain-specific modeling (DSM) is a software development methodology that promises greater gains in productivity by systematizing the use of domain-specific languages (DSL). This paper first addresses the notions of abstraction and specificity by comparing some existing languages, and proposes an original representation that highlights the global advantages of using DSLs. This document presents then MEtaGile, our DSM environment that provides facilities for creating and supporting evolved DSLs. This environment is mainly designed for supporting pragmatic modeling concepts, and implements practical features for supporting the code-generation phase. The development of DSLs is facilitated by the use of a simple but efficient meta-language that allows the domain-specific developers to focus on the final model-to-text transformation; they are neither expected to be expert in modeling nor to master complex transformation languages.

1. Introduction

The software industry is under high pressure to reduce the cost and the development time of the applications, but the global complexity of modern applications increases. In comparison with other engineering branches, such as the automotive industry, the software industry lacks automation, which requires precise models that abstract the system structure and behavior by hiding non-relevant details. The development methods that integrate the models are qualified as Model Driven Development (MDD [1]). The domain field of a general modeling language such as UML is too large for defining precisely the domain-specific concepts needed for the generation, and is mainly used in practice for documentation or discussion [3, 15, 17]. The extension facilities that have appeared with UML 2.0, for supporting MDA [9], make it possible to define specific concepts [5], but UML is already a complex language and many devel-

opers have difficulty to use it efficiently [18]. The domain-specific modeling (DSM [16]) approach, which follows the MDD principles, attempts to reduce the gap between the model and the concrete system by using domain-specific languages (DSL [12]). A DSL provides a specialized semantic, which increases the model precision. In order to be used in practice, a DSL needs a compiler and an adapted CASE tool for helping and guiding the developers during the designing process of the system instances. A DSM environment (Meta-CASE tool or DSL tool) is also needed to simplify the creation of the DSL compiler and of the associated tool. This paper presents MEtaGile, a DSM environment integrated in Eclipse, which provides the required support for defining a DSL and an adapted tool, such as facilities for editing, visualizing and validating the domain-specific models and for automating the generation of end-user systems.

This paper is organized as follows: Section 2 addresses the DSM approach and proposes an original representation of the language properties. Section 3 presents the main features of the MEtaGile environment, and Section 4 addresses the DSL development and presents a simple example.

2. Domain-Specific Modeling

DSM is a MDD approach that represents a viable solution for increasing the productivity of application development [4, 6, 16]. This approach proposes the use of DSLs for modeling systems in a specific domain instead of using general-purpose language (GPL); the DSL model represents at the same time the design, the implementation and the documentation of the system.

An important benefit of DSM results from the splitting of the development process in two successive phases, which can be addressed by different groups of developers. The first phase, the DSL definition must be realized by experts of the domain, and the second phase, the system design can be handled by most developers; the tool is intended to support

the human designing work, and to automate the transformation to concrete code. This approach avoids the stiffness of the traditional development environments; a company can manage its own DSL adapted to its specific needs, and be free to rapidly evolve its definition without external constraints.

2.1. Analysis of the Domain-Specificity

The current section presents an original 2D graph that highlights the benefits of DSLs, and shows how the productivity of a given development is relative to the nature of the language (meta-model) used to develop the system. The global development productivity is positively influenced by the abstraction level (axis x) and by the specificity (axis $-y$) of the language (Figure 1). The abstraction level represents the capacity of a language to hide the concrete details of a system in providing high-level concepts. The specificity level is directly linked to the domain range that a language is able to address. Both notions are not precisely quantifiable, because a language can include different aspects more or less abstract, or specific. However, the global properties of languages can be approached by using 2D areas.

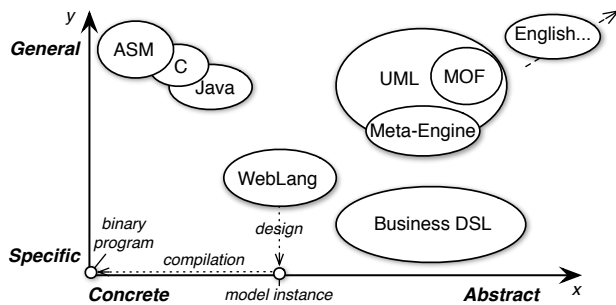


Figure 1. Classification of languages

The presented graph highlights the needed phases for developing an executable system by using different languages. The design phase, represented by the vertical transformation to the bottom axis, is mainly dedicated to the developers, which use the language to create a specific model that implements the system specific requirements. This phase is not automatable along the development of different systems in the same domain. Figure 1 shows that the use of a DSL, close to the business domain of the system, highly reduces the developers intellectual work for creating a specific model instance.

The compilation phase is represented by the horizontal transformation from the specific model instance to the concrete executable code; this process is usually automated by a compiler tool. When a language is highly general and abstract, as is the case of UML, both transformation phases are difficult, and the development of a system is not efficient enough [6, 12].

3. The MEtaGile Environment

The environment we have developed to support the DSM approach provides assistance for creating a new textual DSL or meta-model, adapted to a specific domain (Section 4), and also supports the developers that use the defined DSL to instantiate systems (DSL instance or model).

In comparison with others solutions, such as OAW [14], Microsoft DSL tools [6], GME [11], or MetaEdit+ [19], MEtaGile handles pragmatic development aspects, such as the unique textual model for input, the hierarchical and graphical views for documentation and navigation, the re-definition in real-time of the DSLs and of the templates, or the efficient management of the generated files.

3.1. Extended Textual Modeling

MEtaGile is exclusively based on centralized textual models for editing concerns, which are heavily supported by an evolved generic editor. This editor provides for any DSL models the syntactic highlighting, an editing assist feature, the displaying of the validation messages. It also integrates read-only views, which allow the developers to visualize abstractions of the system, and to navigate efficiently in the textual model. An outline view presents the hierarchical tree of a given model, and diagram views provides graphical representations of the model. MEtaGile enables the presentation of several diagrams that are related with one source model, and thus supports the separation of concerns for documentation and navigation. However, we are convinced that the editing of a complex model is more efficient using a centralized and textual model that fully defines the target system; a precise diagram usually requires the insertion of hidden textual constraints or action language, which are not easily representable using a graphical notation. This centralization enables the simplification of the model-to-text operation, because only one model source is used as input. The use of simple text files also allows the developers to work with the models, and to use efficient and largely adopted textual functionalities, such as the universal *copy/paste* or the CVS sharing; a textual model is also naturally and efficiently editable with the keyboard.

Only the Xtext OAW component also enables the definition of text-to-model transformations, but using MEtaGile the definition of the parser and of the structural concerns of the meta-model are centralized in a unique language. Moreover, our extended BNF syntax is based on JavaCC [7], and allows the handling of more complex expressions, which can also be defined with the Java language.

3.2. Redefinition Capabilities

MEtaGile integrates some features that address specifically the code-generation phase (model-to-text), and of-

fers to the developers a way to support efficiently the successive generations of the target system files, and certain de-synchronization between the model and the produced files. Our code-generation feature uses the template technology JET [8], which is close to the JSP implementation of Apache Tomcat.

3.2.1. Engine Pluggability. MEtaGile is composed of Eclipse plugins that handle the generic features, such as the editor and the views, and of DSL engines, which contain all domain-specific properties. Contrary to the plugins, the modifications applied to the engine are effective in real-time without having to restart the Eclipse platform. This loosely coupled architecture enables the developers to switch easily between different versions of engines, to develop and test in parallel DSL engines and instantiated models, and therefore to increase the development productivity. The other DSM tools are heavier to deploy, and usually require restarting the entire system after a modification. This feature is implemented using a redefined Classloader that addresses the original static classes of the Eclipse plugins and the dynamic classes of the active engine; this dynamic Classloader is used to execute the parsing operation defined in the DSL engine. The generic environment accesses domain-specific information of a model, using the Java reflection and the meta-model defined in the relative DSL engine.

3.2.2. De-Synchronization. When the development addresses the target system details, many manual modifications of the generated system files are often required, and must be preserved. For addressing this purpose MEtaGile enables the developer to visualize and select the replacement mode of the produced files; a report window displays the responsible source model element, the local output path, and the replacement mode. The available modes are *createonly*, *overwrite*, *deactivated* and *merge*; the latter is currently available for the Java files using the JMerge JET technology, and a 3-way merging method is already included for the XML files. This merging technique is simple and flexible enough to support a limited de-synchronization between the model and the output files. The generation mode is stored permanently for each file and can be shared using a CVS server; this feature enables the developers that are not involved in some specific modifications to regenerate the whole system without overwriting some important files or file parts.

3.2.3. Specific Template Redefinition. For addressing the specificities of an application instance, our environment is capable of handling local redefinitions of the templates. This feature offers a flexible way to include application specific properties rapidly and efficiently, in keeping the model and the generated application synchronized. This redefinition is activated by using a specific annotation in front of a model element or globally at the beginning of a model file.

The annotation statement is `@templatedir = package`, where package represents the path where the redefined template class will be emitted. A template file is always relative to a specific node type, and by using similar package identifiers, two instances of the same type can share the same redefined template. MEtaGile provides an operation that loads and prepares the redefined templates into the user project. The application developer can freely modify the content of the templates; the whole model data is accessible using getter functions.

Only OAW currently provides a similar feature, but the redefinition of templates using MEtaGile is in our opinion easier and more flexible. Our templates are modifiable without having an important knowledge about the environment as expected with the OAW approach, and the latter doesn't allow different model elements to use different redefined templates.

4. DSL Engine Modeling

Our solution supports the DSM approach, including the development of the DSL engines using a meta-DSL engine that includes a meta-meta-model. This meta engine allows the domain experts to define and produce a valid DSL engine, which can be used by other developers to define systems in a specific domain. In comparison with other environments, our approach attempts to propose a minimal but simple and centralized way to define and generate the meta-model, the main transformation from text-to-model, and the foundation for the model-to-text transformation, as expected in most cases.

Other approaches usually require defining separately the structural model, the transformations and validation rules, and how these processes are linked together. The definition of a similar DSL tool will require much more effort and knowledge using OAW than using MEtaGile for most cases. Indeed, OAW or other approaches use advanced but heavy languages for specifying transformation and the validation rules, such as QVT, ATL and OCL [13, 10]. Mastering these languages and defining evolved model-to-model transformation and validation rules can improve the quality and the abstraction of the process, but it requires an important investment, and is not adapted with all transformation forms. For instance, when a transformation intends to create output model elements that are not directly related with easily identifiable input model elements, or depends on many different elements, a transformation specified with a Java-like language can be more suitable.

Our meta-DSL, which represents the meta-meta-language of a concrete system, supports bootstrapping [1]. This language is also able to define a full engine component that contains the structure of the meta-model, the basic validation and transformation rules including the parser proper-

ties, which represents the text-to-model process. The structural entities of an engine are specified using a meta-model that includes the principal object-oriented aspects, such as encapsulation, modularity, polymorphism, and inheritance, but it provides specific terms and concept relative to the engine specification. The defined structure is designed for supporting a high modularity; the definition of the checking, processing and producing functions are located in the relative node elements; these entities are qualified as module for main entities and submodule for children elements. Modules and submodules include fields that are intended to specify referenced entities or values. Each node component is susceptible to produce output files, dynamically from a template or by copying a resource file.

MEtaGile also supports the definition of extended graphical views of the model for documentation and navigation concerns; the meta-language includes concepts for creating a view with a selection of modules and sub-fields, and also allows the creation of new elements or sub-element that are not directly related to source model elements (model-to-model transformation). Other DSM environments that support model transformations could also define equivalent views, but the advantage of our approach is to integrate and synchronize the extended views naturally and efficiently in the DSM tool.

4.1. Example of Engine Modeling

This Section presents a realistic example of the definition of a DSL-engine that describes a simple DSL. The target domain of the DSL, presented in the following model, represents a hierarchical web site composed of pages that contain articles and links to other pages. This model defines the various DSL engine properties: the structural elements, the parser syntax and the producing templates. The *page* entity is defined as a main module that includes fields for hosting the relative properties, as the name, the description and the header. The *articles* field list references the articles that are defined locally; an article is defined as a local submodule of the page, and includes simple typed fields, filled by the local parser. The page element includes the list *links* that contains the referenced page names, and the list *pageLinks* that contains the referenced page modules. This last list enables the navigation in the page hierarchy more easily, but it is not automatically filled by the parser and must be populated by the developer in the processing method of the page; this population logic is quite trivial: an iteration of all page links is included in another iteration of all defined page instances, the page is added to *pageLinks* when its name equals the link identifier. Two templates are defined in the page element declaration; the first one is responsible to create an output html file for each page instance, and the second static template attempts to create a unique read-me file that includes a reference on each available page.

```

module Page {
  mainkeyword = "page";
  template = ( page.html, name + ".html");
  templatestatic=(readme.htm,"doc.html","doc");
  String name, head, descr;
  list<Article> articles;
  list<String> links;
  list<Page> pageLinks;
  parser {
    name "{"
      "heading" head = STRING ";";
      ["description" descr = STRING ";";]
      ["links" "=" links ("," links)* ";";]
      ( articles ) *
    "}"
  }
  submodule Article {
    String title, content, authorId;
    boolean isFinished = false;
    int nbWord;
    parser {
      "article" ["finished" isFinished:=true] "{"
        "title" "=" title = STRING ";";
        "content" "=" content = STRING ";";
        ["words" "=" nbWord ";";]
      "}"
    }
  }
}

```

After having generated this specific DSL engine, using the meta-DSL engine, the developer must edit the prepared JET templates and introduce the domain implementation details, here the html code with the dynamic accesses to the page and article properties. Then, system instances can be defined and generated in using the newly generated DSL engine, such as the simple Web-site example specified by the following model.

```

page Index {
  heading "My Watch Company";
  description "Since 1872";
  links = Collections, Sponsoring;
  article finished {
    title = "Happy New Year 2008";
    content = "Our company is happy to...";
    words = 200;
  }
}
page Collections {
  heading "Collections 2008";
  description "The new Collection is...";
  links = Sport, Classic;
  article {
    title = "A specific Watch for...";
    content = "...";
  }
}
page Sport {...} page Classic {...}

```

Figure 2 presents the graphical view output of the given model; each element is displayed as a box, the inner sub-modules are by default folded and displayed in a list, but the user can change these view properties, as the presented *Classic* page, where the articles are presented in a 2D layout. Each element or sub-element that includes a reference on another element is displayed by default in a 2D layout,

and the reference as a link; a link between folded elements is reported to the respective parent and visible element.

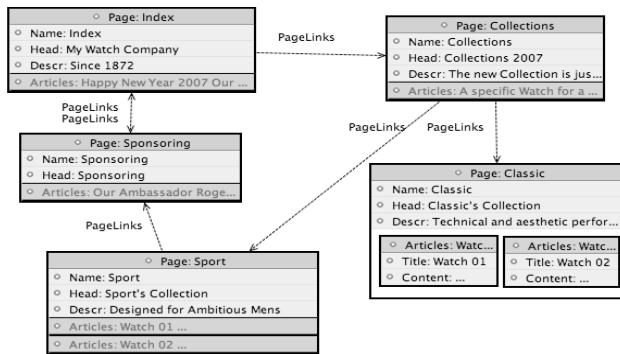


Figure 2. Generic Graphical View

4.1.1. Practical Tests. The previous example is deliberately simple, but the presented meta-DSL is able to specify evolved DSLs as the meta-DSL itself or WebLang [2], which attempts to design the architecture of J2EE Web-applications in a concise and efficient way. It enables the definition of elaborated application models by assembling different component instances, and then producing executable applications, in abstracting the implementation details (see Figure 1). It has been successfully used for three years by hundreds of students of the EPFL for supporting the software engineering course.

Others DSLs have been developed: a Java3D DSL that is able to define evolved 3D scenes in abstracting the framework details, and a PHP DSL able to create evolved sites composed of dynamic pages connected to a database. A DSL, able to define and generate .Net Web sites that includes structural and behavioral concerns, has been developed using the Microsoft DSL tools (2005), and using MEtaGile. This exercise has shown that the Microsoft DSL tools are currently not well adapted to the generation of final applications; the use of MEtaGile and of our textual modeling approach allows the developers to save time in the development of the DSL. The definition of graphical elements requires developing advanced wizards for conducting the developers to set the mandatory properties correctly, but the DSL tools don't offer an efficient support for this task.

5. CONCLUSION

This paper has first discussed abstraction and specificity concerns of some existing languages, and has shown how the combination of these notions influence positively the development productivity. The use of a DSL as a modeling language allows the designers to efficiently manipulate domain-specific concepts; the automation process is also optimized by a precise model and by the specialization of the target domain. The current paper has also presented MEtaGile, a DSM environment, that provides facilities for creating and supporting evolved textual DSLs. This

environment is mainly designed for supporting pragmatic programming, and implements practical features for supporting the code-generation phase; it integrates a loosely coupled architecture that supports rapid DSL evolutions, and a template redefinition functionality, which enables the DSL users to easily adapt some templates for a specific use. The use of a simple but efficient meta-language allows the domain-specific developers to efficiently define textual DSLs; they are not expected to be expert in modeling, and to master transformation and validation languages. Further development will address the definition of DSLs that address more specific business, such as the management of projects, stock, or customers.

References

- [1] C. Atkinson and T. Kuehne. Model-driven development: a metamodeling foundation. *IEEE Software*, Volume 20, Issue 5, Sept.-Oct. 2003 Page(s):36 - 41, 2003.
- [2] O. Buchwalder and C. Petitpierre. WebLang: A Language for Modeling and Implementing Web Applications. In *SEKE06*, 2006.
- [3] M. Fowler. *UML distilled: A brief Guide to the Standard Object Modelling Language*. Object Technology series. Addison Wesley, 3rd edition, 2004.
- [4] M. Fowler. Language workbenches: The killer-app for domain specific languages? www.martinfowler.com, 2005.
- [5] E. Gorshkova and B. Novikov. Exploiting uml extensibility in the design of web applications. <http://citeseer.ist.psu.edu/530237.html>, 2003.
- [6] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03*, pages 16–27. ACM Press, 2003.
- [7] JavaCC. <https://javacc.dev.java.net/>.
- [8] JET. <http://www.eclipse.org/modeling/m2t/?project=jet>.
- [9] A. Kleppe, J. Warmer, and W. Bast. *The Model Driven Architecture-Practice and Promise*. Addison-Wesley, 2003.
- [10] I. Kurtev, K. van den Berg, and F. Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with atl. In *SAC '06*. ACM, 2006.
- [11] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, May 2001.
- [12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM*, 2005.
- [13] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005.
- [14] openArchitectureWare. <http://www.eclipse.org/gmt/oaw>.
- [15] B. Rumpe. Executable modeling with uml. a vision or a nightmare? *Issues and Trends of IT Management*, 2002.
- [16] J.-P. T. Steven Kelly. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE, 2008.
- [17] D. Thomas. Uml - unified or universal modeling language? *Object Technology*, vol. 2, no. 1, January-February, 2003.
- [18] D. Thomas. Mda: Revenge of the modelers or uml utopia? *IEEE Software*, vol. 21, no. 3, pp. 15-17, 2004.
- [19] J.-P. Tolvanen and M. Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *OOPSLA '03*. ACM, 2003.